

# Algoritmi e strutture dati

Koci Erik

April 8, 2021

## 1 Complessità algoritmi

1. **Costo:** si riferisce al costo di un singolo algoritmo
2. **Complessità:** si riferisce a più risoluzioni di un algoritmo

il costo di un blocco **if-then-else** è  $O(\max\{f(n), g(n), h(n)\})$  cioè **O(1)**.

### 1.1 Ordini di grandezza

1.  $\Theta(f(n))$  se cresce tanto quanto f
2.  $O(f(n))$  se la crescita è minore o uguale a f
3.  $\Omega(f(n))$  se la crescita è maggiore o uguale a f

### 1.2 Esercizi

- $1325n^2 + 12n + 1 = \theta(n^3)$  FALSO
- $76n^3 == (n^3)$  VERO
- $n^2 \log n = O(n^2)$  FALSO
- $3^N = O(2^N)$  FALSO
- $1^n = O(2^{\frac{n}{2}})$  FALSO
- $2^N + 100 = O(2^N)$  VERO

- $n = O(n \log n)$  VERO
- $n^2 = (n \log n)$  FALSO
- $\log(n^2) = \Theta(\log n)$  VERO
- $(n + 1)/2 = \Theta(n)$  VERO
- $\frac{(n+1)*n}{2} = \Theta(n^2)$  VERO

### 1.3 Analisi casi

#### 1. Caso pessimo

$$T_{\text{WORST}}(n) = \max T(I)$$

#### 2. Caso ottimo

$$T_{\text{BEST}}(n) = \min T(I)$$

#### 3. Caso medio

$$T_{\text{AVG}}(n) = \sum T(I)P(I)$$

### 1.4 Algoritmi ordinamento

**Selection sort:** scorre tutti gli elementi degli array e si cerca il valore più piccolo scambiando i due valori. Il costo è lineare con il numero di elementi da considerare:

$$T(n) = \Theta(n^2)$$

il costo è  $\Theta(n^2)$  perchè è presente una funzione min che ogni volta controlla se il numero è il minore. Le chiamate a **min** contribuiscono a  $n^2$  mentre il resto combacia con  $n$  cioè  $n^2 + n$ ;  $n$  viene assorbito.

**Ricerca binaria (ricorsiva):** per utilizzare questo algoritmo devo avere un array ordinato. Cerco il valore andando a verificare sempre nella metà dove mi aspetto che sia presente.

$$T(n) = 1 \text{ se } n = 0$$

$$T(n) = T(n/2) + 1$$

**equazione di ricorrenza:** ci aiuta a calcolare il costo analizzando una singola ricorsione.

## 1.5 metodo dell'iterazione:

consiste nello sviluppare l'equazione di ricorrenza, per intuirne la soluzione.

$$T(n) = c_1 + c_2 * \log(n) = \Theta(\log(n))$$

E' presente il **logaritmo** perchè ogni volta devo **dimezzare** il tutto in base al numero di elementi.  $c_1$  perchè devo eseguire le istruzioni la prima volta.

dimostrare che  $T(n) = O(n)$

$$T(n = 1) \quad n == 1$$

$$T(T([n/2]) + n \quad n > 1$$

## 1.6 Metodo della sostituzione:

consiste facendo una dimostrazione per induzione. quindi parto dal valore base che è  $n$  (esempio 1) e dimostriamo che vale anche per un  $n$  più grande.

**caso base:**

$$T(1) = 1 \leq cn$$

**induzione:**

$$\begin{aligned} T(n) &= T([n/2]) + n \\ &\leq c[n/2] + n \quad (\text{ipotesi induttiva}) \\ &\leq cn/2 + n = \frac{cn + 2n}{2} = (c/2 + 1)n \leq cn \end{aligned}$$

## 1.7 Master Theorem:

Si consideri la seguente equazione di ricorrenza:

$$\begin{aligned}T(n) &= d \text{ se } n = 1 \\T(n) &= aT(n/b) + cn^\beta \text{ se } n > 1\end{aligned}$$

e sia:

$$\alpha = \frac{\log(a)}{\log(b)}$$

**a** numero di chiamate ricorsive

**b** mi dice come partiziono il mio input

questi due valori mi danno  $\alpha$ .

$\beta$  mi dice l'esponente che avevo.

L'equazione di ricorrenza ha la seguente soluzione:

1.  $T(n) = \Theta(n^\alpha)$  se  $\alpha > \beta$
2.  $T(n) = \Theta(n^\alpha * \log(n))$  se  $\alpha = \beta$
3.  $T(n) = \Theta(n^\beta)$  se  $\alpha < \beta$

Il teorema fondamentale **non** si può applicare ad algoritmi ricorsivi che non effettuano **partizioni bilanciate**.

ad esempio non può essere applicato nella risoluzione di fibonacci ricorsivo.

Esempio:

$$\begin{aligned}T(n) &= 1 \text{ } n \leq 1 \\T(n1) + T(n2) + 1 &\text{ } n > 2\end{aligned}$$

Se le **partizioni sono bilanciate** conviene utilizzare il **Master Theorem**.

**partizione bilanciate:** quando facciamo chiamate ricorsive prendo il mio input suddividendolo in parti  $n/b$ . Fibonacci non è bilanciato perché abbiamo due chiamate ricorsive diverse.

**L'analisi ammortizzata:** studia il costo medio di una sequenza di operazioni.

Sia  $T(n, k)$  il tempo totale richiesto da un algoritmo, nel caso pessimo, per effettuare  $k$  operazioni su istanze di lunghezza  $n$ . Definiamo il **costo ammortizzato** su una sequenza di  $k$  operazioni come:

$$T_{\alpha}(n) = \frac{T(n, k)}{k}$$

## 1.8 Algoritmi di visita degli alberi

Esistono due tipologie di visita:

- In profondità (pre-ordine, in-ordine, post-ordine)
- In ampiezza

Nella visita **pre-ordine** si parte visitando il nodo della radice per poi passare a visitare tutto il nodo sinistro, risalendo poi andando verso destra.

Nella visita **in-ordine** si parte a visitare dal ramo più in basso a sinistra risalendo per poi andare verso destra.

Nella visita **post-ordine** vengono prima visitati i nodi più in profondità partendo sempre da sinistra verso destra per poi risalire.

Nella visita per **ampiezza** si analizza l'albero a livelli, partendo dalla radice.

## 1.9 Alberi AVL

Un albero *AVL* è un albero di ricerca (quasi) bilanciato. Questo albero supporta le operazioni di *insert()*, *delete()*, *search()* con costo  $O(\log n)$  nel caso pessimo.

### 1.9.1 Fattore di bilanciamento

Il fattore di bilanciamento  $\beta(v)$  di un nodo  $v$  è dato dalla differenza tra l'altezza del sottoalbero sinistro e del sottoalbero destro di  $v$ :

$$\beta(v) = \text{altezza}(v.\text{left}) - \text{altezza}(v.\text{right})$$

### 1.9.2 Bilanciamento in altezza

Un albero si dice **bilanciato in altezza** se le altezze dei sottoalberi sinistro e destro di ogni nodo differiscono al più di uno.

$$\beta \leq 1$$

**Definizione:** un albero *AVL* è un *ABR* bilanciato in altezza.

### 1.9.3 Inserimento e rimozione

Inserimenti e rimozioni richiedono di essere modificati per mantenere il bilanciamento dell'albero.

L'operazione fondamentale per ribilanciare l'albero è la **rotazione semplice**.

### 1.9.4 Rotazione a sinistra

Per effettuare questa rotazione prendo il nodo problematico e effettuo una rotazione scambiandolo con il successivo ed il nodo scambiato diventerà figlio destro mentre il figlio del nodo precedente diventerà figlio sinistro.

## 1.10 Alberi 2-3

Un albero 2-3 è un albero in cui:

- Tutti i percorsi radice-foglia hanno la stessa lunghezza
- Le foglie contengono le chiavi (e i dati da memorizzare) e sono ordinate da sinistra verso destra in ordine di chiave crescente.
- Ogni nodo interno (non foglia)  $v$  ha 2 o 3 figli e mantiene due informazioni
  - $S[v]$ , **chiave massima** nel sottoalbero sinistro (2 o 3 figli)
  - $M[v]$ , **chiave massima** nel sottoalbero centrale (3 figli)
- Distribuzione dei valori  $k$  delle chiavi nei sottoalberi:
  - Sinistro  $k \leq S[v]$
  - Centro  $S[v] < k \leq M[v]$
  - Destro  $k > M[v]$

## 1.11 Tabelle Hash

Le **tabelle hash** hanno una implementazione basata su una chiave  $k$  e array. Per ottenere la chiave sono presenti diverse tecniche di calcolo.

Ricapitolando, per realizzare una tabella hash efficiente abbiamo bisogno di:

- Un vettore
- Una funzione hash calcolabile velocemente e che garantisca una buona distribuzione delle chiavi nel vettore
- Un meccanismo per gestire le collisioni

### 1.11.1 Problema delle collisioni

Una funzione hash  $h$  si dice **perfetta** se è iniettiva:

$$\forall u, v \in U : u \neq v \rightarrow h(u) \neq h(v)$$

Se le collisioni sono inevitabili, è necessario trovare un metodo che le minimizzi, distribuendo **uniformemente** le chiavi negli indici della tabella hash.

### 1.11.2 Funzioni hash

E' necessario fare una premessa; nelle funzioni hash è **sempre possibile** trasformare una chiave complessa in un numero, (conversione in binario).

### 1.11.3 Metodo dell'estrazione

Le caratteristiche di questo metodo sono:

- Usa solo una parte della chiave
- Si seleziona una sottosequenza di  $p$  bit, con  $m = 2^p$
- Solitamente dalle posizioni centrali

**esempio:** Verranno prese le cifre centrali 101000

$$\text{bin}(\text{"beer"}) = 000010\ 000101\ 000101\ 010010$$

- **Vantaggi:** molto veloce da calcolare
- **Svantaggi** rischio collisioni più alto di altri metodi

#### 1.11.4 Metodo della divisione

Basata sul resto della divisione per  $m$ :

- **Vantaggio:** molto veloce
- **Svantaggio:** Suscettibile a specifici valori di  $m$ . Per risolvere questo problema bisogna scegliere  $m$  come numero primo non troppo vicino ad una potenza di 2.

**Esempio:**

$$m = 12, k = 100 \rightarrow h(k) = 4$$

#### 1.11.5 Metodo della moltiplicazione

Basato sulla moltiplicazione e il resto del numero

1. Sia  $A$  una costante,  $0 < A < 1$
2. Moltiplichiamo  $k$  per  $A$  e prendiamo la parte frazionaria
3. Moltiplichiamo quest'ultima per  $m$  e prendiamo la parte intera

**Esempi:**

$$m =, k = 3, A = 0.8 \rightarrow h(k) = 2$$

$$m = 1000, k = 123, A \approx 0.6180339887... \rightarrow h(k) = 18$$

- **Svantaggi:** lento (più lento del metodo di divisione)
- **Vantaggi** Il valore di  $m$  non è critico
- **Come scegliere A?**  $A \approx (\sqrt{5} - 1)/2 = 0.61803... \text{ (Knuth)}$

#### 1.11.6 Metodo della codifica algebrica

Metodo utilizzando dal compilatore java basato su espressioni algebriche:

$$h(k) = (k_n x^n + k_{n-1} x^{n-1} + \dots + k_1 x + k_0) \bmod m$$

$$k = k_n k_{n-1} \dots k_1 k_0$$

Dove  $k_0, k_1 \dots$  possono essere, ad esempio, i bit della **codifica binaria** di  $k$ , oppure i **codici ascii** dei singoli caratteri di  $k$ .

$x$  è un valore **costante**.

- **Vantaggi:** dipende da tutti i bit/caratteri della chiave
- **Svantaggi:**  $n$  addizioni e  $n * (n - 1)/2$  prodotti



### 1.11.7 Problema delle collisioni

Attraverso questi metodi elencati precedentemente siamo riusciti a ridurre il numero di collisioni, ma senza eliminarle.

Per risolvere questo problema la **complessità computazionale potrebbe aumentare a  $n$** , possono essere utilizzate le seguenti tecniche:

1. **Concatenamento**
2. **Indirizzamento aperto**

### 1.11.8 concatenamento

Nella tecnica di **concatenamento** gli elementi con lo stesso valore hash  $h$  vengono memorizzati in una lista concatenata (linked list).

Il **fattore di carico** è dato dal rapporto tra numero di elementi memorizzati e dimensioni della tabella.

La **complessità** del concatenamento è la seguente:

- insert:  $\Theta(1)$
- search:  $\Theta(n)$
- delete:  $\Theta(n)$

### 1.11.9 indirizzamento aperto

L'idea è quella di memorizzare tutte le chiavi nella tabella stessa, ed ogni slot contiene una chiave oppure *null*.

**Inserimento:** se lo slot prescelto è utilizzato, si cerca uno slot alternativo.

**Ricerca:** si cerca nello slot prescelto, e poi negli slot alternativi fino a quando non si trova la chiave oppure *null*.

Vengono utilizzati diversi algoritmi di indirizzamento, per esempio i seguenti:

**Ispezione lineare**

Il primo elemento determina l'intera sequenza. In questo modo si ottengono **lunghe sottosequenze**.

$$h(k, i) = (h'(k) + i)$$

**Ispezione quadratica:**

L'ispezione iniziale è in  $h'(k)$ , mentre le successive hanno un offset che dipende da una **funzione quadratica nel numero di ispezione**.

$$h'(k) + c_1i + c_2i^2$$

**Doppio hashing:**

Formato da **due funzioni ausiliari** di cui la prima  $h_1$  fornisce la prima ispezione, mentre  $h_2$  fornisce l'offset delle successive ispezioni.

$$h(k, i) = (h_1(k) + ih_2(k))$$

**1.11.10 Conclusioni hash table**

Usare funzioni hash  $h(k)$  che producano valori il più possibile uniformemente distribuiti è molto importante perchè altrimenti potremmo arrivare ad una complessità computazionale pari a  $O(n)$ .

**Problemi con hashing:**

- Scarsa locality of reference (cache miss)
- In base all'implementazione è in genere difficile ottenere le chiavi in ordine
- Sebbene il costo medio per operazione sia basso, la singola operazione può risultare molto costosa, ad esempio se occorre ridimensionare la tabella e redistribuire le chiavi.

## 2 Scelta degli algoritmi

A seconda delle operazioni da eseguire è necessario adattare diverse tecniche di implementazione di un algoritmo.

### 2.0.1 Implementazione su un vettore ordinato

Questo tipo di ricerca ha un costo computazionale basso nel caso in cui volessimo **ricercare degli elementi**. Costi computazionali:

- Ricerca  $O(\log n)$
- Inserimento  $O(n)$
- Eliminazione  $O(n)$

### 2.0.2 Implementazione su liste concatenate non ordinate

Questa implementazione converrebbe utilizzarla nel caso in cui volessimo **aggiungere o eliminare degli elementi**. Costi computazionali:

- Ricerca  $O(n)$
- Inserimento  $O(1)$
- Eliminazione  $O(n)$

### 2.0.3 Implementazione alberi ABR

Implementazione basata su alberi binari. Costi computazionali:

- Ricerca  $O(h)$
- Inserimento  $O(h)$
- Eliminazione  $O(h)$

## 2.0.4 Implementazione alberi AVL

Implementazione basata su alberi binari a altezza equivalente. Costi computazionali:

- Ricerca  $O(\log n)$
- Inserimento  $O(\log n)$
- Eliminazione  $O(\log n)$

## 2.0.5 Implementazione alberi 2-3

Implementazione basata su alberi binari ordinati con chiavi massime. Costi computazionali:

- Ricerca  $O(\log n)$
- Inserimento  $O(\log n)$
- Eliminazione  $O(\log n)$

## 2.0.6 Hash table

Implementazione basata su array, dove l'elemento con chiave  $k$  è memorizzato nel  $k -esimo$  "slot" dell'array.

- Ricerca caso medio  $O(1)$                       Ricerca caso pessimo  $O(n)$
- Inserimento caso medio  $O(1)$                       Inserimento caso pessimo  $O(n)$
- Eliminazione caso medio  $O(1)$                       Eliminazione caso pessimo  $O(n)$

## 2.1 Riepilogo

	search		insert		delete	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array ordinato	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista non ordinata	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
ABR	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$
Albero AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Albero 2-3	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Tabella Hash	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$

## 3 Algoritmi di ordinamento

### 3.0.1 ordinamento in loco:

L'algoritmo permuta gli elementi direttamente nell'array originale, senza usare un altro array di appoggio.

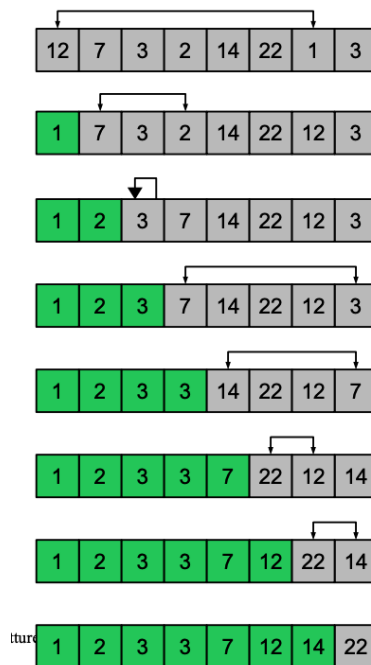
### 3.0.2 ordinamento stabile:

L'algoritmo preserva l'ordine con cui elementi con la stessa chiave compaiono nell'array originale.

## 3.1 Selection sort

Cerca il minimo in  $A[k + 1..n]$  e spostalo in posizione  $k + 1$ .  
La complessità di questo algoritmo è pari a:

$$O(n^2)$$

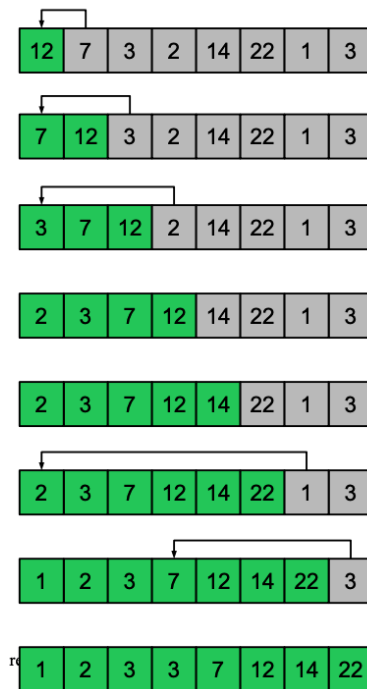


## 3.2 Insertion sort

Inserisco l'elemento di posizione  $k+1$  nella **posizione corretta** all'interno dei primi  $k$  elementi ordinati. Al termine del passo  $k$ , il vettore ha le prime  $k$  componenti ordinate.

Il costo computazionale di questo algoritmo è:

$$O(n^2)$$

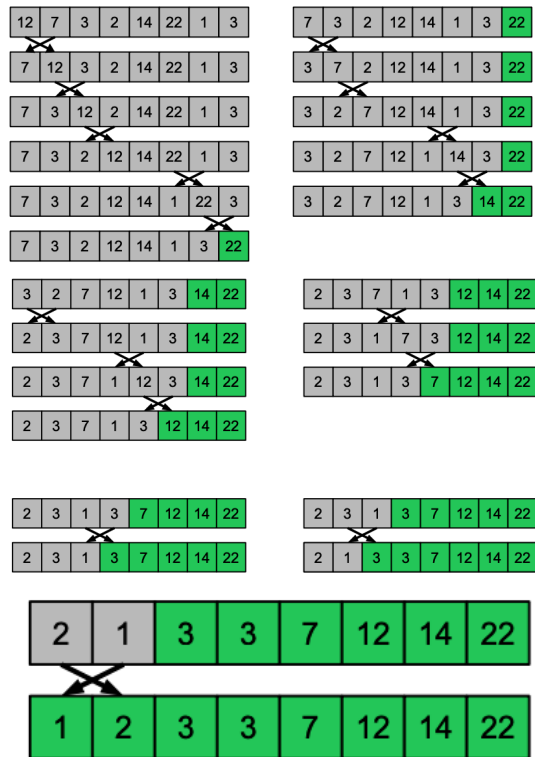


### 3.3 Bubble sort

Ad ogni scansione scambia le coppie di elementi adiacenti che non sono nell'ordine corretto.

Ad ogni scansione **scambia le coppie di elementi adiacenti**. Dopo la prima scansione, l'elemento massimo occupa l'ultima posizione, dopo la k-esima scansione, i k elementi massimi occupano la posizione corretta in fondo all'array. Nel caso *pessimo* – *ottimo* bubble Sort ha costo:

$$\Theta(n^2) \quad \Theta(n)$$



### 3.4 Quick sort

Scegli un elemento  $x$  del vettore  $v$ , e **partiziona il vettore in due parti** considerando gli elementi  $\leq x$  e quelli  $> x$

Ordina ricorsivamente le due parti.

Restituisci il risultato concatenando le due parti ordinate.

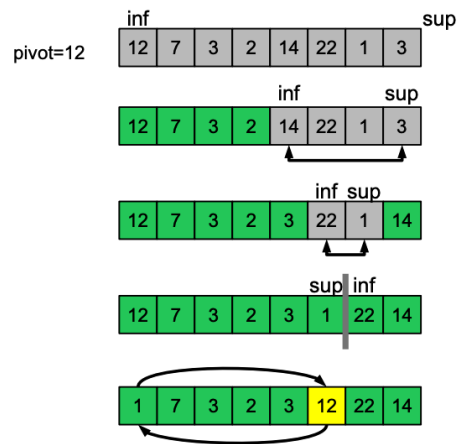
#### 3.4.1 Partizionamento

Manteniamo due indici,  $inf$  e  $sup$ , che vengono fatti **scorrere dalle estremità** del vettore verso il centro. Quando entrambi ( $inf$  e  $sup$ ) non possono essere fatti avanzare verso il centro, si **scambia**  $A[inf]$  e  $A[sup]$ .

Il costo quick sort: Dipende dal partizionamento:

caso peggiore:  $\Theta(n^2)$

caso migliore:  $\Theta(n \log n)$





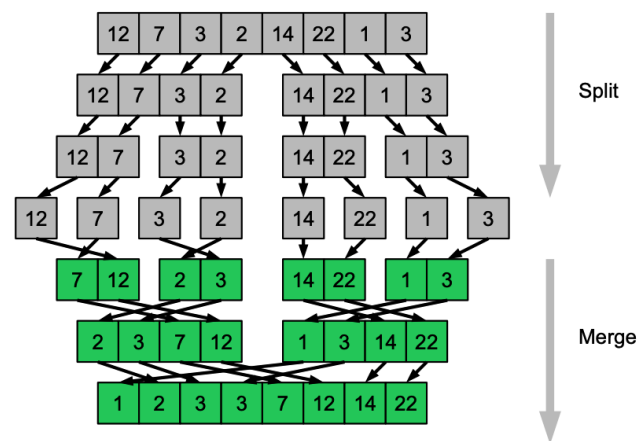
### 3.5 Merge Sort

Questo algoritmo **divide**  $A[]$  in **due meta'**  $A1[]$  e  $A2[]$  (senza permutare) di dimensioni uguali;

Applica **ricorsivamente** Merge Sort a  $A1[]$  e  $A2[]$ .

**Fonde** (merge) gli array ordinati  $A1[]$  e  $A2[]$  per ottenere l'array  $A[]$  ordinato.

#### Merge Sort: esempio



### 3.6 Heapsort

Funzionamento:

1. Costruire un **max-heap** a partire dal vettore  $A[]$  originale, mediante l'operazione **heapify()**
2. Estrarre il **massimo** ( $findMax() + deleteMax()$ )
3. **Inserire** il massimo in ultima posizione di  $A[]$ .
4. **Ripetere** il punto 2. finché lo heap diventa vuoto

### 3.6.1 Albero binario perfetto

Un albero binario è **perfetto** se:

- Tutte le foglie hanno la stessa altezza  $h$
- Nodi interni hanno grado 2

Un albero perfetto ha altezza  $h \simeq \log N$

Il numero di nodi è  $N = \text{nod}i = 2^{h+1} - 1$

### 3.6.2 Albero binario completo

Un albero binario è **completo** se:

- Tutte le foglie hanno profondità  $h$  o  $h-1$
- Tutti i nodi a livello  $h$  sono “accatastati” a sinistra
- Tutti i nodi interni hanno grado 2, eccetto al più uno

### 3.6.3 Max-heap

Un albero binario completo è un albero **max-heap** sse:

- Ad ogni nodo  $i$  viene associato un valore  $A[i]$
- $A[\text{Parent}(i)] \geq A[i]$

### 3.6.4 Min-heap

Un albero binario completo è un albero **min-heap** sse:

- Ad ogni nodo  $i$  viene associato un valore  $A[i]$
- $A[\text{Parent}(i)] \leq A[i]$

## 3.7 Operazioni su array heap

### 3.7.1 findMax()

Individua il valore massimo contenuto in uno heap.

Il massimo è sempre la radice, ossia  $A[1]$ .

L'operazione ha costo  $\Theta(1)$ .

### 3.7.2 fixHeap()

Ripristinare la proprietà di max-heap.

Supponiamo di rimpiazzare la radice  $A[1]$  di un max-heap con un valore qualsiasi, vogliamo fare in modo che  $A[]$  diventi nuovamente uno heap.

### 3.7.3 heapify()

Costruire uno heap a partire da un array privo di alcun ordine.

### 3.7.4 deleteMax()

Rimuovi l'elemento massimo da un maxheap  $A[]$ .

## 4 Selezione del k-esimo

Consideriamo il seguente problema:

**Selezione del k-esimo minimo:** dato un array  $A[1..n]$  di valori distinti e un valore  $1 \leq k \leq n$ , trovare l'elemento che è maggiore di esattamente  $k - 1$  elementi.

**Mediano:** il valore che occuperebbe la posizione  $(n/2)$  se l'array fosse ordinato.

I **motori di ricerca** producono molti risultati a fronte di una singola query. I risultati vengono mostrati in pagine, in ordine decrescente di rilevanza. È **inutile ordinare tutti i risultati** in base alla rilevanza.

Verifichiamo ora i costi computazionali dei singoli casi:

**Ricerca del minimo:**

$$T(n) = n - 1 = \Theta(n)$$

Ricerca del secondo minimo:

$$T(n) = 2n - 3 = \Theta(n)$$

Selezione del k-esimo elemento:

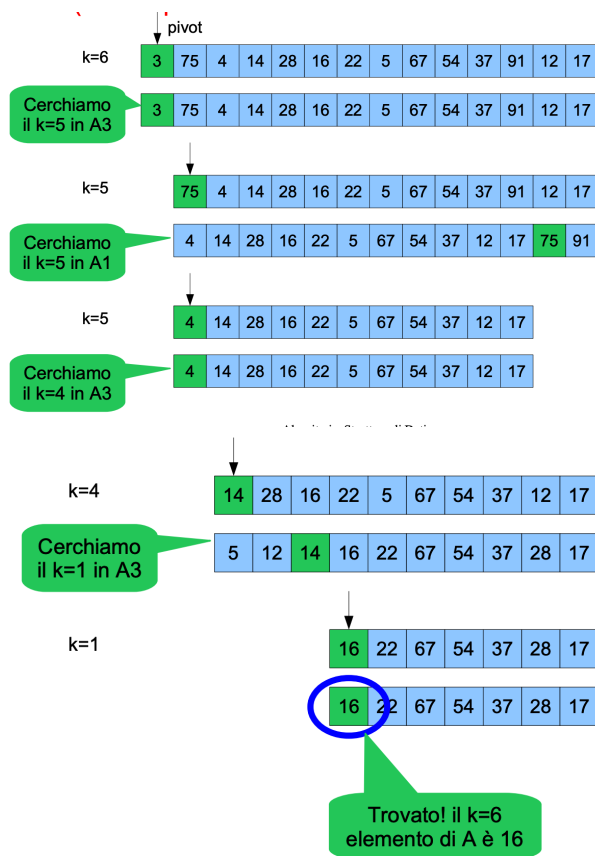
$$T(n) = \Theta(kn)$$

Selezione del valore mediano:

$$T(n) = O(n + k \log n) = O(n + (n/2) \log n) = O(n \log n)$$

#### 4.0.1 Adattamento di quicksort al problema della selezione:

In questo modo divido il mio array in più partizioni, andando a eliminare quelle inutili.



#### 4.0.2 Analisi dell'algoritmo quickSelect()

**Costo nel caso ottimo:**

$$T(n) = T(n/2) + n = \Theta(n)$$

**Costo nel caso pessimo:**

$$T(n) = T(n-1) + n = \Theta(n^2)$$

**Costo nel caso medio:**

$$T(n) \leq 4n$$